Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects

Maurice Herlihy Eric Koskinen
Computer Science Department, Brown University
{mph,ejk}@cs.brown.edu

Abstract

We describe a methodology for transforming a large class of highly-concurrent linearizable objects into highly-concurrent transactional objects. As long as the linearizable implementation satisfies certain regularity properties (informally, that every method has an inverse), we define a simple wrapper for the linearizable implementation that guarantees that concurrent transactions without inherent conflicts can synchronize at the same granularity as the original linearizable implementation.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features – Frameworks; Concurrent programming structures; E.1 [*Data Structures*]: Distributed data structures; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Languages, Theory

Keywords Transactional Boosting, non-blocking algorithms, abstract locks, transactional memory, commutativity

1. Introduction

Software Transactional Memory (STM) has emerged as an alternative to traditional mutual exclusion primitives such as monitors and locks, which scale poorly and do not compose cleanly. In an STM system, programmers organize activities as *transactions*, which are executed atomically: steps of two different transactions do not appear to be interleaved. A transaction may *commit*, making its effects appear to take place atomically, or it may *abort*, making its effects appear not to have taken place at all.

To our knowledge, all transactional memory systems, both hardware and software, synchronize on the basis of *read/write conflicts*. As a transaction executes, it records the locations (or objects) it read in a *read set*, and the memory locations (or objects) it wrote in a *write set*. Two transactions *conflict* if one transaction's read or write set intersects the other's write set. Conflicting transactions cannot both commit. Conflict *detection* can be eager (detected before it occurs) or lazy (detected afterwards). Conflict *resolution* (deciding which transactions to abort) can be implemented in a variety of ways.

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.

Synchronizing via read/write conflicts has one substantial advantage: it can be done automatically without programmer participation. It also has a substantial disadvantage: it can severely and unnecessarily restrict concurrency for certain shared objects. If these objects are subject to high levels of contention (that is, they are "hot-spots"), then the performance of the system as a whole may suffer.

Here is a simple example. Consider a mutable set of integers that provides $\mathsf{add}(x)$, $\mathsf{remove}(x)$ and $\mathsf{contains}(x)$ methods with the obvious meanings. Suppose we implement the set as a sorted linked list in the usual way. Each list node has two fields, an integer value and a node reference next. List nodes are sorted by value, and values are not duplicated. Integer x is in the set if and only if a list node has value field x. The $\mathsf{add}(x)$ method reads along the list until it encounters the largest value less than x. Assuming x is absent, it creates a node to hold x, and links that node into the list.

Consider a set whose state is $\{1, 3, 5\}$. Transaction A is about to add 2 to the set and transaction B is about to add 4. Since neither transaction's pending method call depends on the other's, there is no inherent reason why they cannot run concurrently. Nevertheless, calls to $\mathsf{add}(2)$ and $\mathsf{add}(4)$ do have read/write conflicts in the list implementation, because no matter how A and B's steps are interleaved, one must write to a node read by the other. Unlike conflicts between short-term locks, where the delay is typically bounded by a statically-defined critical section, if transaction A is blocked by B, then A is blocked while B completes an arbitrarily long sequence of steps.

By contrast, a high level of concurrency can be realized in a lock-based list implementation such as *lock coupling* [2]. All critical sections are short-lived, and multiple threads can traverse the list concurrently. Moreover, there also exist well-known *lock-free* list implementations [20] that provide even more fine-grained concurrency, relying only on individual compareAndSet() calls for synchronization.

Several "escape" mechanisms have been proposed to address the limitations of STM concurrency control based on read/write conflicts. For example, *open nested transactions* [22] (discussed in more detail later) permit a transaction to commit the effects of certain nested transactions while the parent transaction is still running. Unfortunately, lock-coupling's critical sections do not correspond naturally to properly-nested sub-transactions. Lock coupling can be emulated using an *early release* mechanism that allows a transaction to drop designated locations from its read set [13], but it is difficult to specify precisely when early release can be used safely, and the technique seems to have limited applicability. We are not aware of any prior escape mechanism that approaches the level of concurrency provided by common lock-free data structures.

We are left in the uncomfortable position that well-known and efficient data structures can easily be made concurrent in standard non-transactional models, but appear to be inherently sequential in

all known STM models. If transactional synchronization is to gain wide acceptance, however, it must support roughly the the same level of concurrency as state-of-the-art lock-based and lock-free algorithms, among transactions without real data dependencies. This challenge has two levels: transaction-level and thread-level. At the coarse-grained, transactional level, a transaction adding 2 to the set should not have to wait until a transaction adding 4 to the same set completes. Equally important, at the fine-grained thread level, the concurrent calls should be able to execute at the same degree of interleaving as the best existing lock-based or lock-free algorithms.

This paper introduces *transactional boosting*, a methodology for transforming a large class of highly-concurrent linearizable objects into highly-concurrent transactional objects. We describe how to transform a highly-concurrent linearizable *base object*, implemented without any notion of transactions, into an equally concurrent transactional object.

Transactional boosting treats each base object as a *black box*. It requires only that the object provide a *specification* characterizing its *abstract state* (for example, it is a set of integers), and how its methods affect the state (for example, $\mathsf{add}(x)$ ensures that x is in the set). Transactional boosting also requires certain regularity conditions (basically, that methods have inverses) which we will discuss later.

Transactional boosting complements, but does not completely replace conventional read/write synchronization and recovery. We envision using boosting to implement libraries of highly-concurrent transactional objects that might be synchronization hot-spots, while ad-hoc user code can be protected by conventional means.

This paper makes the following contributions:

- To the best of our knowledge, transactional boosting is the first STM technique that relies on object *semantics* to determine conflict and recovery.
- Because linearizable base objects are treated as black boxes, transactional boosting allows STMs to exploit the considerable work and ingenuity that has gone into libraries such as `java.util.concurrent`.
- Because we provide a precise characterization of how to use the technique correctly, transactional boosting avoids the deadlock and information leakage pitfalls that arise in open nested transactions [22].
- We identify and formally characterize an important class of *disposable* method calls whose properties can be exploited to provide novel transactional approaches to semaphores, reference counts, free-storage management, explicit privatization, and related problems.
- Preliminary experimental evidence suggests that transactional boosting performs well on simple benchmarks, primarily because it performs both conflict detection and logging at the granularity of entire method calls, not individual memory accesses. Moreover, the number of aborted transactions (and wasted work) is substantially lower.

It must be emphasized that all of the mechanisms we deploy originate, in one form or another, in the database literature from the 70s and 80s. Our contribution is to adapt these techniques to software transactional memory, providing more effective solutions to important STM problems than prior proposals.

2. Software Transactional Memory

We assume an STM where transactions can be serialized in the order they commit, a property called *dynamic atomicity* [30]. For brevity, we assume for now that transactions are not nested. We require the ability to register user-defined handlers called when transactions commit or abort (as provided by DSTM2 [12] and SXM [1]). We now describe our methodology in more detail, postponing formal definitions to Section 5.

Any transactional object must solve two tasks: *synchronization* and *recovery*. Synchronization requires detecting when transactions conflict, and recovery requires discarding speculative changes when a transaction aborts.

The specification for a linearizable base object defines an *abstract state* (such as a set of integers), and a *concrete state* (such as a linked list). Each method is usually specified by a *precondition* (describing the object's abstract state before invoking the method) and a *postcondition*, describing the object's abstract state afterwards, as well as the method's return value.

Informally, two method invocations *commute* if applying them in either order leaves the object in the same state and returns the same response. In a Set, for example, $\mathsf{add}(x)$ commutes with $\mathsf{add}(y)$ if x and y are distinct. This commutativity property is the basis of how transactional boosting performs conflict detection.

We define an *abstract lock* [22] associated with each invocation of a boosted object. Two abstract locks *conflict* if their invocations do not commute. Abstract locks ensure that non-commutative method calls never occur concurrently. Before a transaction calls a method, it must acquire that method's abstract lock. The caller is delayed while any other transaction holds a conflicting lock (timeouts avoid deadlock). Once it acquires the lock, the transaction makes the call, relying on the base linearizable object implementation to take care of thread-level synchronization. In the integer set example, the abstract locks for $\mathsf{add}(x)$ and $\mathsf{add}(y)$ do not conflict when x and y are distinct, so these calls can proceed in parallel.

A method call m has *inverse* m' if applying m' immediately after m undoes the effects of m. For example, a method call that adds x to a set not containing x has as inverse the method call that removes x from the set. A method call that adds x to a set already containing x has a trivial inverse, since the set's state is unchanged.

When inverses are known, recovery can be done at the granularity of method calls. As a transaction executes, it logs an inverse for each method call in a thread-local log. If the transaction commits, the log is discarded, and the transaction's locks are released. However, if the transaction aborts, the transaction revisits the log entries in reverse order executing each inverse. (A transaction that added x to the set would call $\mathsf{remove}(x)$.) When every inverse has been executed, the transaction releases its locks.

Sometimes it is convenient to delay certain method calls until after a transaction commits or aborts. For example, consider an object that generates unique IDs for transactions. The object's abstract state is the pool of unused IDs. It provides an $\mathsf{assignID}()$ method that returns an ID not currently in use, removing it from the pool, and a $\mathsf{releaseID}(x)$ method that returns ID x to the pool. Any two $\mathsf{assignID}()$ calls that return distinct IDs commute, and a $\mathsf{releaseID}(x)$ call commutes with every call except an $\mathsf{assignID}()$ call that returns x. As a result, if a transaction that obtains x aborts, we can postpone returning x to the pool for arbitrarily long, perhaps forever. For example, if the ID generator is implemented as a counter, then it is sensible never to return x to the pool. We call these *disposable* method calls.

There are other examples of disposable methods. One can implement a *transactional semaphore* that decrements a counter immediately, blocking while the counter value is zero, but postpones incrementing the counter until the calling transaction commits. *Reference counts* would follow a dual strategy: the reference count is incremented immediately, but decremented lazily after the transaction commits. (When an object's reference count is zero, its space can be freed.) Reference counter decrements can also be postponed, allowing deallocation to be done in batches. Similar disposability

tradeoffs apply to transactional malloc() and free(), and counters used to manage "privatization" of objects shared by transactional and non-transactional threads.

A boosted object can also be accessed outside of a transaction, as long as the thread acquires the appropriate abstract locks. Accessing a boosted object outside of a transaction does not prevent other transactions from accessing the same object. However, abstract locks ensure that all transactional operations which do not commute with the non-transactional operations are delayed until the non-transactional thread releases the abstract lock. By contrast, external access is difficult in traditional STM implementations because non-transactional threads modify memory without acquiring locks, and their effects cannot be aborted. This is precisely the *privatization* problem discussed in [28].

Transactional boosting is not a panacea. It is limited to objects (1) whose abstract semantics are known, (2) where commutative method calls can be identified, and (3) for which reasonably efficient inverses either exist or can be composed from existing methods. This methodology seems particularly well suited to collection classes, because it is usually easy to identify inverses (for example, the inverse of removing x is to put it back), and many method calls commute (for example, adding or removing x commutes with adding or removing y, for $x \neq y$).

Further, transactional boosting supports a clean separation between low-level thread synchronization, which is the responsibility of the underlying linearizable object implementation, and high-level transactional synchronization, which is handled by the abstract locks and undo log. Non-conflicting concurrent transactions synchronize at the level of the linearizable base object, implying for example, that if the base object is non-blocking for concurrent threads, then it is non-blocking for concurrent non-conflicting transactions. No prior STM technique can achieve this kind of fine-grained thread-level parallelism.

Set Specification

Method	Inverse
add(x)/*false*	noop()
add(x)/*true*	remove(x)/*true*
remove(x)/*false*	noop()
remove(x)/*true*	add(x)/*true*
contains(x)/_	noop()

Commutativity

insert (x)/_ $\Leftrightarrow$ insert (y)/_, $x \neq y$
remove(x)/_ $\Leftrightarrow$ remove(y)/_, $x \neq y$
insert (x)/_ $\Leftrightarrow$ remove(y)/_, $x \neq y$
add(x)/*false* $\Leftrightarrow$ remove(x)/*false* $\Leftrightarrow$ contains(x)/_

Figure 1. Specification of a Set

```
1  public class SkipListKey {
2    ConcurrentSkipListSet<Integer> list;
3    LockKey lock;
4    ...
5    public boolean add(final int v) {
6      lock.lock(v);
7      boolean result = list.add(v);
8      if (result) {
9        Thread.onAbort(new Runnable() {
10         public void run() { list.remove(v);}}
11         );
12     }
13     return result;
14   }
15   ...
16 }
```

Figure 2. The SkipListKey class

3. Examples

We now consider some examples illustrating how highly-concurrent linearizable data structures can be adapted to provide the same fine-grained thread-level concurrency in transactional systems. Our presentation is informal, postponing more precise definitions to Section 5.

For each example we provide a specification, such as that of the Set in Figure 1. We use the notation method(v)/r to indicate the invocation of method with argument v and response r. In some cases the response is inconsequential to commutativity and is denoted _ and void method calls are simply denoted method(v). Finally, we use the infix symbol $\Leftrightarrow$ to mean that its two arguments commute, and $\not\Leftrightarrow$ when they do not.

3.1 Sets

A Set is a collection of *items* without duplicates. Like the integer set described above, a Set provides add(x), remove(x), and contains(x) methods. A call to add() or remove() returns a Boolean indicating whether the set was modified. Each of these calls has an inverse. A remove(x) call that returns *true* has inverse add(x), and vice-versa. Note that the inverse of a call often depends on its result: the inverse to a remove(x) call that returns *false* is vacuous.

Many calls commute: either order yields the same results and produces the same final state. For example, consider a set in state $\{5, 16, 29\}$. If we call add(3) and remove(29) in either order, both return *true*, and the set ends up in state $\{3, 5, 16\}$. Transactional boosting allows transactions to call commutative methods without blocking, independently of how the set is actually implemented. If either transaction aborts, the inverse method calls will be applied, undoing the aborted transaction's changes to the Set's abstract state.

Figure 1 summarizes Set methods' inverses and commutativity. Indeed *any* call to add(x), remove(y), or contains(z) commutes with the others so long as they have distinct arguments.

Skip List Implementation A *skip list* [23] is linked list in which each node has a set of short-cut references to later nodes in the list. A skip list is an attractive way to implement a Set because it provides logarithmic-time add(), remove(), and contains() methods.

To illustrate our claim that we can treat base linearizable objects as black boxes, we describe how to transactionally-boost the ConcurrentSkipListSet class from the java.util.concurrent library. This class is a very efficient, but complicated, lock-free skip list. We will show how to transform this highly-concurrent linearizable library class into an equally concurrent transactional library class without the need to understand how the linearizable object is implemented.

Figure 2 shows part of the SkipListKey class, a transactional Set implementation that is constructed by boosting the ConcurrentSkipListSet object using a LockKey for synchronization. For brevity, we focus on implementing a set of integers, called *keys*. Before we describe the implementation of the boosted ConcurrentSkipListSet class, we consider some utility classes.

The LockKey class, as shown in Figure 3, associates an abstract lock with each key. Key-based locking may block commutative calls (for example, two calls to add(x) when x is in the set), but it provides enough concurrency for practical purposes. (Naturally, transactional boosting does not require the programmer to exploit all commutative methods.) This class's commit and abort handlers release the locks (on abort, after replaying the log). The lock's

```
17 public class LockKey {
18   ConcurrentHashMap<Integer,Lock> map;
19   public LockKey() {
20     map = new ConcurrentHashMap<Integer,Lock>();
21   }
22   public void lock(int key) {
23     Lock lock = map.get(key);
24     if (lock == null) {
25       Lock newLock = new ReentrantLock();
26       Lock oldLock = map.putIfAbsent(key, newLock);
27       lock = (oldLock == null) ? newLock : oldLock;
28     }
29     if (LockSet.add(lock)) {
30       if (!lock.tryLock(LOCK_TIMEOUT,
31        TimeUnit.MILLISECONDS);) {
32         lockSet.remove(lock);
33         Thread.getTransaction().abort();
34         throw new AbortedException();
35       }
36     }
37   }
38   ...
39 }
```

Figure 3. The LockKey class

map field is a ConcurrentHashMap (Line 18) that maps integers to locks.[1] The lock(k) method first checks whether there exists a lock for this key, and if not, creates one (Lines 23-28).

Each transaction has a thread-local LockSet tracking the locks it has acquired that must be released when the transaction commits or aborts. The transaction must register commit and abort handlers instructing the STM to release all locks (after replaying the log, if necessary). The transaction tests whether it already has that lock (Line 29). If so, nothing more needs to be done. Otherwise, it tries to acquire the lock (Line 31). If the lock attempt times out, it aborts the transaction (Lines 31-35).

In the boosted skip list shown in Figure 2, an add(v) call first acquires the lock for v (Line 6), and then calls the linearizable base object's add(v) method (Line 7). If the return value indicates that the base object's state has changed, then the caller registers an *abort handler* to call the inverse method (Line 8). All acquired abstract locks are automatically released when the transaction commits or aborts.

3.2 Priority Queues

A *priority queue* (PQueue) is a collection of keys, where the domain of keys has a natural total order. Unlike Sets, PQueues may include duplicate keys. A priority queue provides an add(x) method that adds x to the collection, a removeMin() method that returns and removes the *least* key in the collection, and a min() method that returns but does not remove the least key.

Priority queue methods and their inverses are listed in Figure 4. The inverse for a removeMin() call that returns x is just add(x). In most linearizable heap implementations, removing x and adding it again may cause the internal structure of the heap to be restructured, but such changes do not cause synchronization conflicts because the PQueue's abstract set is unchanged. The min() method does not change the queue's state, and needs no inverse.

Most priority queue classes do not provide an inverse to add(x). Nevertheless, it is relatively easy to synthesize one. We create a simple Holder class containing the key and a Boolean deleted field, initially *false*. Holders are ordered by their key values. Instead of adding the key to the PQueue, we add its holder. To undo the effects of an add() call, the transaction sets that Holder's deleted field to *true*, leaving the Holder in the queue. We change the transactional removeMin() method to discard any deleted records returned by the linearizable base object's removeMin(). (We will show an example later.)

[1] We use maps and locks from the Java concurrency packages.

Priority Queue Specification	
Method	**Inverse**
removeMin()/x	add(x)/_
min()/x	noop()
add(x)/_	addInverse(x)/_

Commutativity

add(x)/_ $\Leftrightarrow$ add(y)/_

removeMin()/x $\Leftrightarrow$ add(y)/_, $x \leq y$

min()/x $\Leftrightarrow$ min()/x

Figure 4. Specification of a Priority Queue

```
40 public class HeapRW {
41   ConcurrentHeap<Holder> heap;
42   LockRW lock;
43   public void add(int item) {
44     lock.readLock();
45     heap.add(new Holder(item));
46     Thread.onAbort(new Runnable() {
47       public void run() {holder.deleted = true;}
48       });
49   }
50   public int removeMin() {
51     Holder holder;
52     lock.writeLock();
53     do {
54       holder = heap.removeMin();
55     } while (holder.deleted);
56     if (holder != null) {
57       Thread.onAbort(new Runnable() {
58         public void run() { heap.add(holder); }
59       });
60     }
61     return holder.value;
62   }
```

Figure 5. The HeapRW class

All add() calls commute. Additionally, removeMin()/x commutes with add(y) if $x \leq y$. Here too, commutativity depends on both method call arguments and results.

Heap Implementation Priority queues are often implemented as *heaps*, which are binary trees where each item in the tree is less than its descendants. We implemented the linearizable concurrent heap implementation due to Hunt *et al.* [16]. The removeMin() method removes the root and re-balances the tree, while add(x) places the new value at a leaf, and then "percolates" the value up the tree. This implementation uses fine-grained locks. (Because locks are not nested, this algorithm is not a good candidate for open nested transactions.)

Figure 5 shows part of the boosted heap implementation. The heap field (Line 41) is the base linearizable heap, and the lock field (Line 42) is a two-phase readers-writers lock. The readLock() method acquires the lock in shared mode, and writeLock() in exclusive mode. All such locks are released when the transaction commits or aborts. Each add() call acquires a shared-mode lock (Line 44), relying on the base object's thread-level synchronization to coordinate concurrent add() calls. As described earlier, the

add() method does not add the key directly to the base heap, but instead creates a Holder containing the key and a Boolean deleted flag (Line 45). For recovery, it logs a call to mark that key's Holder as deleted (Line 46).

```
63 public class BlockingQueue<T> {
64   BlockingDeque<T> queue;
65   TSemaphore full;   // block if full
66   TSemaphore empty; // block if empty
67   public BlockingQueue(int capacity) {
68     queue = new LinkedBlockingDeque<T>(capacity);
69     full  = new TSemaphore(capacity);
70     empty = new TSemaphore(0);
71   }
72   public void offer(final T value) {
73     full.acquire();
74     queue.offerLast(value);
75     empty.release();
76     Thread.onAbort(new Runnable() {
77       public void run() {queue.takeLast();}
78     };
79   }
80   public T take() {
81     empty.acquire();
82     T result = queue.takeFirst();
83     full.release();
84     Thread.onAbort(new Runnable() {
85       public void run() { queue.offerFirst(result); }
86     };
87     return result;
88   }
89 }
```

Figure 6. The BlockingQueue class

3.3 Pipelining

Pipelining is a well-established way to achieve concurrency in certain applications. Algorithms employing pipelining are common in areas such as networking or graphics. In typical pipelined applications, one thread (or transaction) is in charge of each stage of the pipeline. Because different pipeline stages may need varying amounts of time, threads communicate by bounded queues called *buffers*.

Transactional pipelining introduces the need for *conditional synchronization*: a transaction that encounters an empty (or full) buffer should block until that buffer becomes non-empty (or non-full). The need for conditional synchronization implies that not all existing STMs support pipelining.

We now consider the BlockingQueue methods and their inverses. Following Java conventions, the BlockingQueue class provides two methods: offer() enqueues a work item on a queue, while take() dequeues a work item. If the queue is full, however, offer() blocks until there is room, and if the queue is empty, take() blocks until a work item is available. Because BlockingQueue does not provide inverses, we take as linearizable base class a blocking *double-ended queue* (BlockingDEQueue) taken from java.util.concurrent. This class provides the methods offerFirst(x), offerLast(x), takeFirst(), and takeLast(). A transactional offer(x) call results in a linearizable offerFirst(x) call, with inverse takeFirst().

Because BlockingQueue objects are shared by pairs of transactions, of which one repeatedly calls offer(), and the other repeatedly calls take(), we care only about commutativity between these two methods. Here, commutativity depends on the queue's abstract state: offer() commutes with take() if and only if the buffer is non-empty. The full pipeline specification is given in the Technical Report [11].

Unique ID Generator Specification

Method	Inverse	Post-Abort
assignID()/x	noop()	releaseID(x)/_

Commutativity

assignID()/x ⇔ assignID()/y $x \neq y$

assignID()/x ⇔ assignID()/x

Figure 7. Specification of a Unique ID Generator

Pipeline Implementation To detect when BlockingQueue methods within a pipeline can proceed in parallel, we introduce a *transactional semaphore* class (TSemaphore) to mirror the queue's committed state. Figure 6 shows the BlockingQueue implementation. It uses two transactional semaphores: the full semaphore blocks the caller when the queue is full by counting the number of empty slots. It is initially set to the queue capacity (Line 69). The empty semaphore blocks the caller while the queue is empty by counting the number of items in the queue. It is initially set to zero (Line 70). As noted above, the acquire() method, which decrements the semaphore, takes effect immediately, blocking the caller while the semaphore's committed state is zero. The release() method is disposable: it takes effect only when the transaction commits. We discuss another example of disposable methods in the next subsection. Note also that transactional semaphores cannot be implemented by conventional read/write synchronization: they require boosting to avoid deadlock.

The offer() method decrements the full semaphore before calling the base queue's offerLast() method (Line 73). When the decrement returns, there is room. After placing the item in the base queue, offer() increments the empty semaphore (Line 75), ensuring that the item will become available after the transaction commits. The take() method increments and decrements the semaphores in the opposite order.

3.4 Unique Identifiers

Generating unique IDs is a well-known problem for STMs based on read/write conflicts. The obvious approach, incrementing a shared counter, introduces false read/write conflicts. Under transactional boosting, we would define an ID generator class that provides an assignID() method that returns an ID distinct from any other ID currently in use. Note that assignID()/x commutes with assignID()/y for $x \neq y$.

If a thread aborts after obtaining ID x from assignID() then, strictly speaking, we should put x back by calling releaseID(x), which returns x to the pool of unused IDs. Nevertheless, release is disposable: we can postpone putting x back (perhaps forever). As long as x is assigned, no transaction can observe (by calling assignID()) whether x is in use. Figure 7 shows the commutativity specification for a unique ID generator. Transactional boosting not only permits a transactional unique ID generator to be implemented as a getAndAdd() counter, it provides a precise explanation as to why this implementation is correct.

4. Evaluation

We now describe some experiments testing the performance of transactional boosting.

Stanford STAMP Benchmarks We modified two of the Stanford *STAMP* benchmarks [5] (written in C) to use boosting. The vacation benchmark simulates a travel reservation system in which client threads interact with a database consisting of a collection of red-black trees. In our transactionally-boosted red-black tree implementation, methods synchronize by short-term mutual exclusion locks, and each key is assigned its own two-phase trans-

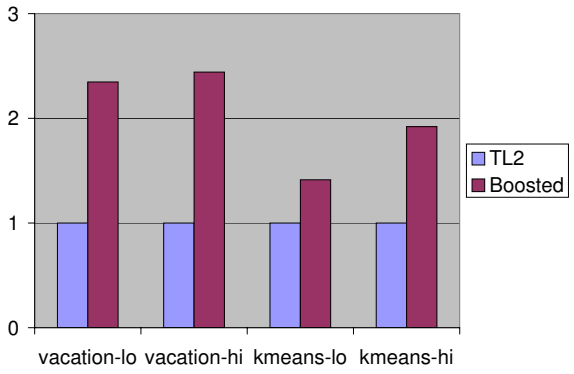

Figure 8. Throughput for boosted STAMP benchmarks normalized against the throughput of conventional TL2, which maintains shadow copies.

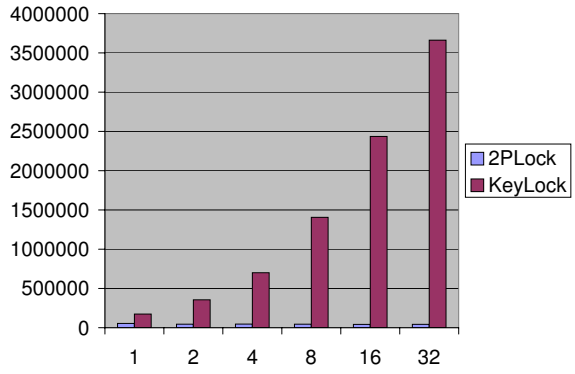

Figure 9. Throughput for transactionally-boosted skip lists using simple (left) and key-based (right) two-phase locks.

actional lock by hashing into an array of locks. Both long and short-term locks are test-and-test-and-set spin locks.

The kmeans benchmark assigns objects to one of k clusters based on a similarity function. Adding an object to one cluster commutes with adding an object to a different cluster. We changed fewer than 15 lines of code to boost this algorithm, using short-term mutual exclusion locks and two-phase transactional locks.

Figure 8 shows the normalized throughput of four STAMP benchmark tests: vacation with low and high contention[2] and kmeans with low and high contention[3]. These tests were run on a multiprocessor with four 2.0 GHz Xeon processors, each one two-way hyper-threaded, for a total of eight threads. In all four tests, boosting substantially improved throughput relative to the baseline TL2 [7] implementation (normalized here to 1).

Lock-Free Skip List The remaining tests were implemented in Java using the DSTM2 [12] software transactional memory system and the java.util.concurrent libraries. These experiments were run on a Sun Microsystems T2000 system with 32 cores.

To test the effect of transactional lock granularity, we tested two boosted implementations of the lock-free ConcurrentSkipListSet base class. The first uses a single transactional lock for all method calls, while the second uses a lock per key. Because they use the same base object, difference in throughput can be attributed entirely to differences in parallelism. Transactions insert and remove random keys from disjoint ranges. Figure 9 shows that fine-grained transactional synchronization can have a dramatic effect on throughput, especially when combined with fine-grained thread-level synchronization.

Concurrent Heap Figure 10 shows the relative throughput of two heap implementations executing half add() calls and half removeMin() calls. As noted above, the base object is protected by a readers/writers lock, where add() calls acquire the shared reader's lock, and removeMin() calls the exclusive writer's lock. This experiment suggests that using a read/write lock to discriminate between add() and removeMin() calls is worthwhile.

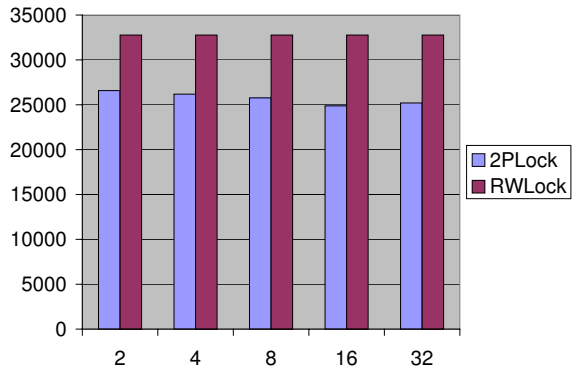

Figure 10. Throughput for boosted heap implementations using an exclusive lock (left) and a shared/exclusive lock (right).

5. Formal Model

We now summarize the formal model. A full discussion, complete with examples, can be found in the Technical Report [11]. This model is adapted from Weihl [30] and from Herlihy and Wing [15].

5.1 Histories

A computation is modeled as a *history*, that is, a sequence of instantaneous *events*. Events associated with changes in the status of a transaction T include $\langle T\ \mathsf{init}\ \rangle$, $\langle T\ \mathsf{commit}\rangle$, $\langle T\ \mathsf{abort}\rangle$ indicating that T starts rolling back its effects, and $\langle T\ \mathsf{aborted}\rangle$ indicating that T finishes rolling back its effects. Additionally, the event $\langle T$, $\mathsf{x.m}(v)\ \rangle$ indicates that T invokes method m of object x with argu-

[2] We used the following switches, which STAMP recommends for low and high contention contention, respectively. See [5] for the semantics of each switch.
Low switches: −n4 −q90 −u80 −r65536 −t4194304
High switches: −n8 −q10 −u80 −r65536 −t4194304

[3] Low switches: −m40 −n40 −t0.05 −i inputs/random1000_12
High switches: −m20 −n20 −t0.05 −i inputs/random1000_12

ment v, and the event $\langle T,\ v\ \rangle$ indicates the corresponding return value v. For example, here is a history in which transaction A adds 3 to a skip list:

$$\langle A\ \mathsf{init}\ \rangle \cdot \langle A,\ \mathsf{list.add}(3)\ \rangle \cdot \langle A, true\ \rangle \cdot \langle A\ \mathsf{commit}\rangle$$

A single transaction run in isolation defines a *sequential history*. A *sequential specification* for an object defines a set of *legal* sequential histories for that object. A *concurrent history* is one in which events of different transactions are interleaved.

A *subhistory* denoted $h|T$ is a subsequence of the events of h, restricted to a transaction T. Two histories h and h' are *equivalent* if for every transaction A, $h|A = h'|A$. If h is a history, $committed(h)$ is the subsequence of h consisting of all events of committed transactions.

Definition 5.1. *A history h is* strictly serializable *if $committed(h)$ is equivalent to a legal history in which these transactions execute sequentially in the order they commit.*

Definition 5.2. *Histories h and h'* define the same state *if, for every history g, $h \cdot g$ is legal if and only if $h' \cdot g$ is.*

Definition 5.3. *For a history h and any given invocation I and response R, let I^{-1} and R^{-1} be the* inverse *invocation and response. That is, the invocation and response such that the state reached after the history $h \cdot I \cdot R \cdot I^{-1} \cdot R^{-1}$ is the same as the state reached after history h.*

In the Skip List example, if an element is added to a list and then removed, the list is returned to its initial state. For this example, remove() is the inverse of insert () (eliding item values for the moment).

If I does not modify the data structure, its inverse I^{-1} is trivial; we denote it noop(). Note that inverses are defined in terms of method calls (that is, invocation/response pairs), not invocations alone. For example, one cannot define an inverse for the removeMin() method call of a heap without knowing which value it removed.

Definition 5.4. *Two method calls I, R and I', R'* commute *if: for all histories h, if $h \cdot I \cdot R$ and $h \cdot I' \cdot R'$ are both legal, then $h \cdot I \cdot R \cdot I' \cdot R'$ and $h \cdot I' \cdot R' \cdot I \cdot R$ are both legal and define the same state.*

Commutativity identifies method calls that are in some sense orthogonal and have no dependencies on each other. In the Skip List example, we can take advantage of the commutativity of the insert () method for distinct values. No matter how these method calls are ordered, they leave the object in the same final state.

For a history h, let $\mathcal{G}$ be the set of histories g such that $h \cdot g$ is legal.

Definition 5.5. *A method call denoted $I \cdot R$ is* disposable *if, $\forall g \in \mathcal{G}$, if $h \cdot I \cdot R$ and $h \cdot g \cdot I \cdot R$ are legal, then $h \cdot I \cdot R \cdot g$ and $h \cdot g \cdot I \cdot R$ are legal and both define the same state.*

In other words, the method call $I \cdot R$ can be postponed arbitrarily without anyone being able to tell that $I \cdot R$ did not occur. When $I \cdot R$ does occur it may alter future computation, but postponing it still results in a legal history. In the above definition we quantify over all possible histories that proceed h and end with $I \cdot R$.

In the Unique ID Generator example, a transaction may delay the release of an identifier until after it commits. Since releaseID () is a disposable method, regardless of how long it is postponed, computation yields legal histories; other transactions acquire alternate identifiers.

5.2 Rules and Correctness

A transactional boosting system must follow these rules.

1. **Commutativity Isolation.** For any non-commutative method calls $I_1, R_1 \in T_1$ and $I_2, R_2 \in T_2$, either T_1 commits or aborts before any additional method calls in T_2 are invoked, or vice-versa.

   Informally, commutativity isolation stipulates that methods which are not commutative can be executed, so long as they are not executed concurrently. Note that this rule does not specify a locking discipline, but rather specifies a property of histories resulting from all possible (correct) disciplines. In practice, choosing a locking discipline is an engineering decision. A discipline that is optimal in the sense that no two commutative operations are serialized may suffer performance overhead from the computation involved in implementing the locking discipline. By contrast, an overly conservative approximation may inhibit all concurrency. In Section 4 we quantified this trade-off with some examples.

2. **Compensating Actions.** For any history h and transaction T, if $\langle T\ \mathsf{aborted}\rangle \in h$, then it must be the case that $h|T = \langle T\ \mathsf{init}\ \rangle \cdot I_0 \cdot R_0 \cdots I_i \cdot R_i \cdot \langle T\ \mathsf{aborted}\rangle \cdot I_i^{-1} \cdot R_i^{-1} \cdots I_0^{-1} \cdot R_0^{-1} \cdot \langle T\ \mathsf{aborted}\rangle$ where i indexes the last successfully completed method call.

   This rule concerns the behavior of an aborting transaction. At the point when a transaction decides to abort, it must subsequently invoke the inverse method calls of all method calls completed thus far. The transaction need not acquire locks to undo its effects. This property is important because for alternative techniques, such as nested open transactions, care must be taken to ensure that compensating actions (the analog of inverse methods) do not deadlock.

3. **Disposable Methods** For any history h and transaction T, any method call $\langle T,\ \mathsf{xm}(v)\ \rangle \cdot \langle T,\ r\ \rangle$ that occurs after $\langle T\ \mathsf{commit}\rangle$ or after $\langle T\ \mathsf{abort}\rangle$ must be disposable.

   As a result of this rule, if T generates a method call after it commits, regardless of how far into the future the method call occurs, the history is legal. The timing of these delayed disposable methods is an engineering decision, as discussed in Section 3.

Theorem 5.1. *(Main Theorem) For any STM system that obeys the correctness rules, the history of committed transactions is strictly serializable.*

All proofs are available in [11].

6. Related Work

Transactional memory [14] has gained momentum as an alternative to locks in concurrent programming. This approach has been investigated in hardware [14, 21], in software [8, 9, 13, 27], and in schemes that mix hardware and software [6, 25]. Existing STMs synchronized via read/write conflicts, which may cause *benign* conflicts (see Harris et al [10]). Here, we describe how to synchronize in a way that exploits object *semantics*.

Open nested transactions [22] (ONT) have been proposed as a way to implement highly-concurrent transactional objects. In ONT, a nested transaction can be designated as *open*. If an open transaction commits, its effects immediately become visible to all other transactions. The programmer can register handlers to be executed when transactions enclosing the open transaction commit or abort.

Although transactional boosting and ONT both use abstract method-based locks, the two approaches are starkly divergent. Open nested transactions are a *mechanism*, not a methodology. By themselves, they provide no guidance as to how they can be used correctly. As noted by Ni *et al.* [22], care must be taken to avoid deadlocks in abort handlers, and to avoid unexpected behavior that may occur if an open nested transaction's read set intersects a parent's write set. In transactional boosting, however, inverse methods called by an aborting transaction cannot deadlock with other ongoing transactions because the aborting transaction acquires no additional locks. Moreover, because the base object's methods are not called in a nested transaction, read/write conflicts between parents and open children do not arise.

When comparing boosting to ONT, it is important to distinguish between different kinds of potential deadlocks. Like McRT [25], transactional boosting uses two-phase locking, which is vulnerable to deadlock, but can be avoided by timeouts. Deadlocks on lock acquisition are qualitatively different from the deadlocks that arise in ONT because it is possible to recover from lock acquisition deadlock by aborting and retrying a transaction that times out. By contrast, in ONT, there is no way to recover if an aborting transaction deadlocks while executing its abort handler.

Open nested transactions also have certain limitations in expressive power. It is unclear how to map open nested transactions onto algorithms that use techniques such as lock coupling, where synchronization regions are not properly nested. Moreover, because transactions enforce isolation, there is no possibility of thread-level concurrency between open nested transactions, and therefore no way to exploit existing thread-level synchronization libraries. Finally, using open nested transactions to construct, say, a highly-concurrent transactional hash table, requires re-implementing the hash table itself, while transactional boosting would treat the hash table as a black box.

Harris and Stipić [10] recently proposed "abstract nested transactions" (ANTs). Although ANTs also aim to reduce benign conflicts, the approach is substantially different from ours. Unlike our methodology, memory access during an abstract nested transaction is logged, and this log is used to detect conflicts. As a commit-time optimization, ANT re-evaluates expressions (closures) which are part of conflicting data access to ensure the expression value has not changed since it was first computed. While this approach is well-suited to mostly-functional languages, it is unclear how well it could be used in a language where re-evaluating closures may have side effects. Transactional boosting does not need to track memory access as an executing transaction will not conflict with any other.

Many of the mechanisms used by transactional boosting are well-known from other contexts. Other work on commutativity-based synchronization includes Bernstein [4], Steele [29], Diniz and Rinard [24], Weihl [30], Schwartz and Spector [26], Beeri [3], and Korth [17].

Transactional boosting also benefits from logging high-level method calls, instead of low-level memory accesses, a notion introduced by Lomet, and by Schwarz and Spector [19, 26].

Kulkarni *et al.* [18] describe Galios, a system that exploits commutativity and inverses for efficient thread-level speculation. Moravan *et al.* [21] and Zilles and Baugh [31] observe that constructs similar to open nested transactions can be used to allow transactions to execute non-transactional code, such as system calls.

7. Conclusion

We have presented a methodology for translating a large class of highly-concurrent linearizable objects into highly-concurrent transactional objects. Given only a well-defined specification of a black-box object, transactional boosting allows concurrent threads to interact with the object within a transaction, and no log of memory access is needed neither for conflict detection nor for recovery from aborted transactions. We have shown that for many workloads, the additional run-time burden of transactional boosting is far offset by the performance gain of eliminating memory access logging. Finally, our approach guarantees that the history of computation remains strictly serializable in the presence of arbitrarily many concurrent transactions and abortions.

We defer one matter to future work. Deadlock is possible if two threads attempt to acquire two of the same abstract locks in opposite order. One possible solution is to introduce a timeout when a thread is attempting to acquire a lock; when the timeout expires, the thread can partially abort itself with the hope of releasing the lock that another thread is attempting to acquire.

There are many ways in which transactional boosting can be extended. It could encompass STMs based on nested transactions using techniques similar to those employed by LogTM [21]. Transactions could be extended to encompass multiple threads, using abstract locks for transactional synchronization, and relying on the base object for thread-level synchronization. In a hybrid system that combines small hardware transactions with STM, one could implement base method calls as hardware transactions, using boosting to managed long-lived software transactions.

Acknowledgments

We are grateful to Tim Harris for pointing out that transactional boosting can be extended to support the `retry()` construct for conditional synchronization from STMHaskell [9], monitoring lock releases to reschedule retried transactions. This work was partially supported by grants from Sun Microsystems and Intel Corporation.

References

[1] SXM 1.1: Software Transactional Memory package for C#. http://research.microsoft.com/research/downloads/Details/6cfc842d-1c16-4739-afaf-edb35f544384/Details.aspx.

[2] BAYER, R., AND SCHKOLNICK, M. Concurrency of operations on B-Trees. *Acta Informatica 9* (1977), 1–21.

[3] BEERI, C., BERNSTEIN, P., GOODMAN, N., LAI, M.-Y., AND SHASHA, D. A concurrency control theory for nested transactions (preliminary report). In *Proceedings of the 2nd annual ACM symposium on Principles of distributed computing (PODC '83)* (New York, NY, USA, 1983), ACM Press, pp. 45–62.

[4] BERNSTEIN, A. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers 15*, 5 (1966), 757–763.

[5] CAO MINH, C., TRAUTMANN, M., CHUNG, J., MCDONALD, A., BRONSON, N., CASPER, J., KOZYRAKIS, C., AND OLUKOTUN, K. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA 07)*. Jun 2007.

[6] DAMRON, P., FEDOROVA, A., LEV, Y., LUCHANGCO, V., MOIR, M., AND NUSSBAUM, D. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII)* (New York, NY, USA, 2006), ACM Press, pp. 336–346.

[7] DICE, D., SHALEV, O., AND SHAVIT, N. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC '06)* (2006), pp. 194–208.

[8] HARRIS, T., AND FRASER, K. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA '03)* (2003), ACM Press, pp. 388–402.

[9] HARRIS, T., MARLOW, S., PEYTON-JONES, S., AND HERLIHY, M. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN symposium on Principles and practice of parallel*

programming (PPoPP '05) (New York, NY, USA, 2005), ACM Press, pp. 48–60.

[10] HARRIS, T., AND STIPIĆ, S. Abstract nested transactions. http://research.microsoft.com/ tharris/papers/2007-ant.pdf, 2007.

[11] HERLIHY, M., AND KOSKINEN, E. Transactional boosting: A methodology for highly-concurrent transactional objects. Tech. Rep. CS-07-08, Brown University, Department of Computer Science, 2007.

[12] HERLIHY, M., LUCHANGCO, V., AND MOIR, M. A flexible framework for implementing software transactional memory. In *Proceedings of the 21th ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA '06)* (2006), pp. 253–262.

[13] HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER, III, W. N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd annual symposium on Principles of distributed computing (PODC '03)* (2003), ACM Press, pp. 92–101.

[14] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)* (1993), ACM Press, pp. 289–300.

[15] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[16] HUNT, G. C., MICHAEL, M. M., PARTHASARATHY, S., AND SCOTT, M. L. An efficient algorithm for concurrent priority queue heaps. *Information Processing Letters 60*, 3 (1996), 151–157.

[17] KORTH, H. F. Locking primitives in a database system. *J. ACM 30*, 1 (1983), 55–79.

[18] KULKARNI, M., PINGALI, K., WALTER, B., RAMANARAYANAN, G., BALA, K., AND CHEW, P. Optimistic parallelism requires abstractions. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '07)* (2007).

[19] LOMET, D. B. MLR: a recovery method for multi-level systems. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, jun 1992), vol. 21, ACM Press, pp. 185–194.

[20] MICHAEL, M. M. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '02)* (2002), ACM Press, pp. 73–82.

[21] MORAVAN, M. J., BOBBA, J., MOORE, K. E., YEN, L., HILL, M. D., LIBLIT, B., SWIFT, M. M., AND WOOD, D. A. Supporting nested transactional memory in LogTM. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)* (New York, NY, USA, 2006), ACM Press, pp. 359–370.

[22] NI, Y., MENON, V. S., ADL-TABATABAI, A.-R., HOSKING, A. L., HUDSON, R. L., MOSS, J. E. B., SAHA, B., AND SHPEISMAN, T. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '07)* (New York, NY, USA, 2007), ACM Press, pp. 68–78.

[23] PUGH, W. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures* (1989), pp. 437–449.

[24] RINARD, M. C., AND DINIZ, P. C. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems 19*, 6 (November 1997), 942–991.

[25] SAHA, B., ADL-TABATABAI, A.-R., HUDSON, R., MINH, C. C., AND HERTZBERG, B. Mcrt-STM. In *Proceedings of the 11th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '06)* (2006).

[26] SCHWARZ, P. M., AND SPECTOR, A. Z. Synchronizing shared abstract types. *ACM Transactions on Computer Systems 2*, 3 (1984), 223–250.

[27] SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)* (1995), ACM Press, pp. 204–213.

[28] SPEAR, M. F., MARATHE, V. J., DALESSANDRO, L., AND SCOTT, M. L. Privatization techniques for software transactional memory. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC '07)* (New York, NY, USA, 2007), ACM, pp. 338–339.

[29] STEELE, JR, G. L. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '90)* (New York, NY, USA, 1990), ACM Press, pp. 218–231.

[30] WEIHL, W. Data-dependent concurrency control and recovery (extended abstract). In *Proceedings of the second annual ACM symposium on Principles of distributed computing (PODC '83)* (New York, NY, USA, 1983), ACM Press, pp. 63–75.

[31] ZILLES, C., AND BAUGH, L. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT '06)*. June 2006.